平成19年度オペレーティングシステム期末試験
2008年2月4日

問題は 3 問，6 ページある．

1

大きさ N バイトのファイル F を読み込み, そのチェックサムを計算するプログラムを考える. チェックサムの計算方法は色々あるが, ここでは単純にデータの各バイトを整数とみなしてその和をとることとする. これを行うプログラムの一例を書けば以下のようになる.

```
1:  char checksum_of_file() {
2:      int fd = open(F, O_RDONLY);
3:      char s = 0;
4:      int i;
5:      char * a = malloc(N);
6:      read(fd, a, N);
7:      for (i = 0; i < N; i++) {
8:          s += a[i];
9:      }
10:     return s;
11: }
```

このプログラムを 20 回連続して繰り返し実行 (同じファイル F を 20 回読む) して, 一回あたりの実行時間 (checksum_of_file の開始から終了まで) を測定する. 一回あたりの実行時間は繰り返し 20 回実行したうちの最後の 10 回の実行時間の平均をとる. プログラムは PC やサーバ用の, 現代的な OS (Unix や Windows) 上で実行し, このプロセス以外に測定を邪魔するようなアクティブに実行しているプロセスはないものとする.

(1) 様々な N に対して実行時間はどのようにかわるか? 横軸に N, 縦軸に測定された実行時間をとったグラフを書け.

(2) なぜ実行時間がそのように振る舞うのかを説明せよ. 特に, グラフの重要な量 (傾きや, 形が不連続に変化する点など) について, それがコンピュータのどのようなパラメータで決まるのかに言及せよ.

(3) ファイルを読むのに malloc と read を用いる代わりにファイルマッピングを用いるとどのようになるか. (1), (2) と同様の問いに答えよ. グラフは malloc と read を用いた場合との違いが分かるように, (1) の結果に重ねて書け. そしてその違いはなぜ生ずるのかについて説明せよ. Unix では, プログラムの 5, 6 行目を以下に入れ替えることで, ファイルマッピングを用いてファイルを読むことができる.

```
char * a = mmap(0, N, PROT_READ, MAP_PRIVATE, fd, 0);
```

2

1 つのスレッド c (以下の関数 C を実行) が, 二つのスレッド p (以下の関数 P を実行), q (以下の関数 Q を実行) のどちらかからデータを受け取り, 受け取ったデータを用いて計算を行うプログラムを書きたい. そのために共有メモリ X に p が, Y に q がデータを書き込み, c が X や Y を読むことにする. c は p, q の**どちらかでもデータを書き込んだらそれを読み込んで実行を始める**こととする.

簡単のため X, Y は int 型で, 3 スレッドが実行を開始する前, X, Y には 0 が入っており, p, q が書き込む値は決して 0 ではないとする. 全体として以下のようなプログラムになり, 関数 P, Q, C が別々のスレッドによって実行される.

```
volatile int X = 0;
volatile int Y = 0;
P(arg) {
  ...
  X = ...;
}
Q(arg) {
  ...
  Y = ...;
}
C(arg) {
  int t;
  (*) X または Y が 0 でなくなるまで待ちそれを t に読み込む;
  t に読み込んだ値を使って計算をする;
}
```

(1) 「(∗)X または Y が 0 でなくなるまで待ちそれを t に読み込む」部分は単純には以下のように実現できる.

```
/* X または Y が 0 でなくなるまで待ちそれを t に読み込む; ... (*) */
int t;
while (1) {
  t = X;
  if (t != 0) break;
  t = Y;
  if (t != 0) break;
}
```

これは一応正しく動くものの問題があり, 通常は推奨されない. どのような問題か.

(2) 上で述べた問題を解決するために OS やスレッドライブラリが提供するセマフォを使う方法がある. この方法にしたがって上記のプログラムを修正せよ.

(3) セマフォを用いずに条件変数と排他制御を用いて同じ事を行え (実質的には条件変数と排他制御でセマフォを実装できることを示している).

付録として, (2), (3) に関連する API のマニュアルページ (抜粋) をつける.

セマフォ関係:

```
SYNOPSIS
   int sem_init(sem_t *sem, int pshared, unsigned int value);
   int sem_wait(sem_t *sem);
   int sem_post(sem_t *sem);

DESCRIPTION
   sem_init() initializes the unnamed semaphore at the address pointed to by sem.
   The value argument specifies the initial value for the semaphore.

   The pshared argument indicates whether this semaphore is to be shared between
   the threads of a process, or between processes.

   If pshared has the value 0, then the semaphore is shared between the threads of
   a process, and should be located at some address that is visible to all threads
   (e.g., a global variable, or a variable allocated dynamically on the heap).

   sem_wait() decrements (locks) the semaphore pointed to by sem.  If the
   semaphore's value is greater than zero, then the decrement proceeds, and
   the function returns, immediately.  If the semaphore currently has the
   value zero, then the call blocks until either it becomes possible to
   perform the decrement (i.e., the semaphore value rises above zero), or a
   signal handler interrupts the call.

   sem_post() increments (unlocks) the semaphore pointed to by sem.  If the
   semaphore's value consequently becomes greater than zero, then another
   process or thread blocked in a sem_wait(3) call will be woken up and
   proceed to lock the semaphore.
```

条件変数関係:

```
SYNOPSIS
   int pthread_cond_init(pthread_cond_t *restrict cond,
                         const pthread_condattr_t *restrict attr);
   int pthread_cond_broadcast(pthread_cond_t *cond);
   int pthread_cond_wait(pthread_cond_t *restrict cond,
                         pthread_mutex_t *restrict mutex);

DESCRIPTION
   The pthread_cond_init()  function shall initialize the condition variable
   referenced by cond with attributes referenced by attr. If attr  is NULL,
   the  default  condition  variable  attributes shall be used; the effect is
   the same as passing the address of a default condition  variable
   attributes  object.  Upon successful initialization, the state of the
   condition variable shall become initialized.

   The pthread_cond_broadcast() function shall unblock all threads currently
   blocked on the specified condition variable cond.


   The pthread_cond_wait() function shall block on a condition variable. It shall
   be called with mutex locked by the calling thread or undefined behavior
   results.


   It atomically releases mutex and causes the calling thread to block on the
   condition variable cond; atomically here means "atomically with respect to
   access by another thread to the mutex and then the condition
   variable". That is, if another thread is able to acquire the mutex after
   the about-to-block thread has released it, then a subsequent call to
   pthread_cond_broadcast() or pthread_cond_signal() in that thread shall behave as
   if it were issued after the about-to-block thread has blocked.
```

```
   Upon successful return, the mutex shall have been locked and shall be owned by
   the calling thread.

   When using condition variables there is always a Boolean predicate involving
   shared variables associated with each condition wait that is true if the
   thread should proceed.  Spurious wakeups from the pthread_cond_wait() function
   may occur.  Since the return from pthread_cond_wait() does not imply
   anything about the value of this predicate, the predicate should be re-evaluated
   upon such return.
```

排他制御関係:

```
SYNOPSIS
   int pthread_mutex_init(pthread_mutex_t *restrict mutex,
                          const pthread_mutexattr_t *restrict attr);
   int pthread_mutex_lock(pthread_mutex_t *mutex);
   int pthread_mutex_unlock(pthread_mutex_t *mutex);

   The pthread_mutex_init() function shall initialize the mutex
   referenced by mutex with attributes specified by attr.  If attr is
   NULL, the default mutex attributes are used; the effect shall be the
   same as passing the address of a default mutex attributes
   object. Upon successful initialization, the state of the mutex
   becomes initialized and unlocked.

   The mutex object referenced by mutex shall be locked by calling
   pthread_mutex_lock().  If the mutex is already locked, the calling
   thread shall block until the mutex becomes available.  This operation
   shall return with the mutex object referenced by mutex in the locked
   state with the calling thread as its owner.

   The pthread_mutex_unlock() function shall release the mutex object
   referenced by mutex.  The manner in which a mutex is released is
   dependent upon the mutex's type attribute.  If there are threads
   blocked on the mutex object referenced by mutex when
   pthread_mutex_unlock() is called, resulting in the mutex becoming
   available, the scheduling policy shall determine which thread shall
   acquire the mutex.
```

3

OS のスレッド (プロセス) スケジューラに関する以下の会話を読んで, その後の問いに答えよ.

加藤: 今日 OS の授業でスレッドのスケジューリングについて習ったんだけど, 俺には何が難しいのかよくわからなかったな.

松尾: おうそらどうして?

加藤: 全部のスレッドを一定時間ずつ, 代わりばんこに実行させる. 以上. これ以上何が必要なの?

松尾: それですんだら楽だよな. いったいそのマシンにはいくつのスレッドがあると思う?

(加藤は (1) プロセスのリストを表示するコマンド を実行させて数える)

加藤: ざっと見て, プロセスが 50 個はあるな. ということは少なくとも 50 のスレッドがあるということか. こんなに自分で立ち上げた覚えはないんだけど...

松尾: お前の言う通りだとすると, お前が演習で作ったソーティングのプログラムも, CPU 本来の速度の 1/50 くらいで動いていたのか?

加藤: もしそうだったら... **悔しいです!**

松尾: 演習のプログラムを走らせる時にはエディタ, ブラウザ, メールソフトを全部止めなきゃいけないとしたら不便だし, 第一もったいないよな.

加藤: そうか, 思い出した. そういえばスレッドには [(2)] という状態があって, その状態のスレッドには決して CPU が割り当てられないんだった.

松尾: そう, そしてさっきの 50 個のプロセスのほとんどは [(2)] だったというわけ.

加藤: でも, だとしたら [(2)] でない状態, つまり [(3)] 状態のスレッドに一定時間代わりばんこに実行させる. 以上. これじゃダメ?

松尾: 代わりばんこというが, (4) スレッドが実行をしている途中に [(2)] になる ことがあるよね. そういう場合その実行時間がどんなに短くでも代わりばんこ, つまり他のスレッドが一回ずつ実行するまで, そのスレッドは実行されないわけ? もしお前がそのスレッドだったらどう思う?

加藤: うーん... **悔しいです!**

松尾: だよな.

加藤: そうか, 思い出した. スレッドが「どのくらい CPU 時間を使ったか」をもう少しこまめに把握してあげればいいんだ. 各スレッドに一定時間の「貯金」をあげて, 実行時間にしたがって減らしていき, 0 になったら次の奴に切り替える. 0 になる前に [(2)] になった人は, まだ貯金が残っているから, もう一度 [(3)] 状態に戻ったら実行を再開できる. 皆に貯金を同じだけ与えれば公平だ. できた!

松尾: ひとたび貯金が 0 になったスレッドはどうなる? いつかまた貯金をあげないといけないね.

加藤: 全スレッド, いや, 全部の [(3)] 状態のスレッドの貯金が 0 になったら, また同じ額だけの貯金をあげればいいんじゃないかな.

松尾: そのとき, [(2)]のスレッドにはまだ貯金が残っているかもしれない. それらのスレッドは, どうする?

加藤: 民主主義の国では, やはり他のスレッドと同じだけあげるのがいいと思います. つまり, そのとき存在しているスレッドすべてに, 一定量の貯金を与えます. コードで書けば,

for すべてのスレッド t:
  t の貯金 += 一定値...(∗)

松尾: 確かにそうかもしれないが, 深刻な問題がある. さっき, ほとんどのスレッドは [(2)] であると言ったよね. だとすると, [(5)]. これはあまり望ましいとは言えないよな.

加藤: うーん... **悔しいです!** じゃこうします. さっきの「そのとき存在しているスレッドすべてに一定額の貯金を与える」かわりに, 「そのとき存在しているスレッドすべてを, 一定額の貯金にリセットする」. つまり, 貯金は一度すべて剥奪してその後で一定額の貯金を渡します. コードで書けば,

for すべてのスレッド t:
  t の貯金 = 一定値 ...(∗∗)

松尾: なるほどね. いわゆる「単年度会計」ってやつで, 余った額はすべて政府に返しなさいというやつね. だいぶいいんだけどもうひとひねりするともっと良くなる. それは公平で効率的というだけでなく, スケジューラのもう一つの目標である「対話的プロセスの応答性の向上」に寄与するアイデアなんだけど.

加藤: そうか, 思い出した. [] コードで書けばこんな感じです.

[(6)]

松尾: そう. 実はそれが実質的には Linux 2.4 スケジューラのやっていることなんだ.

加藤: なるほどねー. で, お前いつからそんなに OS に詳しくなったんだ?

松尾: いやー, ○○○ ちゃんに OS 教えてって言われちゃって, 必死で勉強したんだよ. おかげで今はラブラブなんだよ〜〜(**注:** あまり世の中にそういうことはない)

加藤: **く, 悔しいです!**

(1) 下線部 (1) について, そのようなコマンド・プログラムの例を具体例をあげよ.

(2) [(2)] に適切な言葉を入れよ.

(3) [(3)] に適切な言葉を入れよ.

(4) 下線部 (4) について, どのような場合にそうなるか, いくつかの例を示せ.

(5) [(5)] には直前で加藤が提案しているスケジューラの深刻な問題を説明する文章が入る. 考えて作文せよ.

(6) [(6)] に入るべき適切なコードの例を書け. それは (∗) や (∗∗) と同じくらい簡潔なコードであり, (∗) にある深刻な問題を持たず, 対話的プロセスの応答性の向上に役に立つ. また, そのコードがそれらの性質を持つ理由もあわせて書け.

問題は以上である